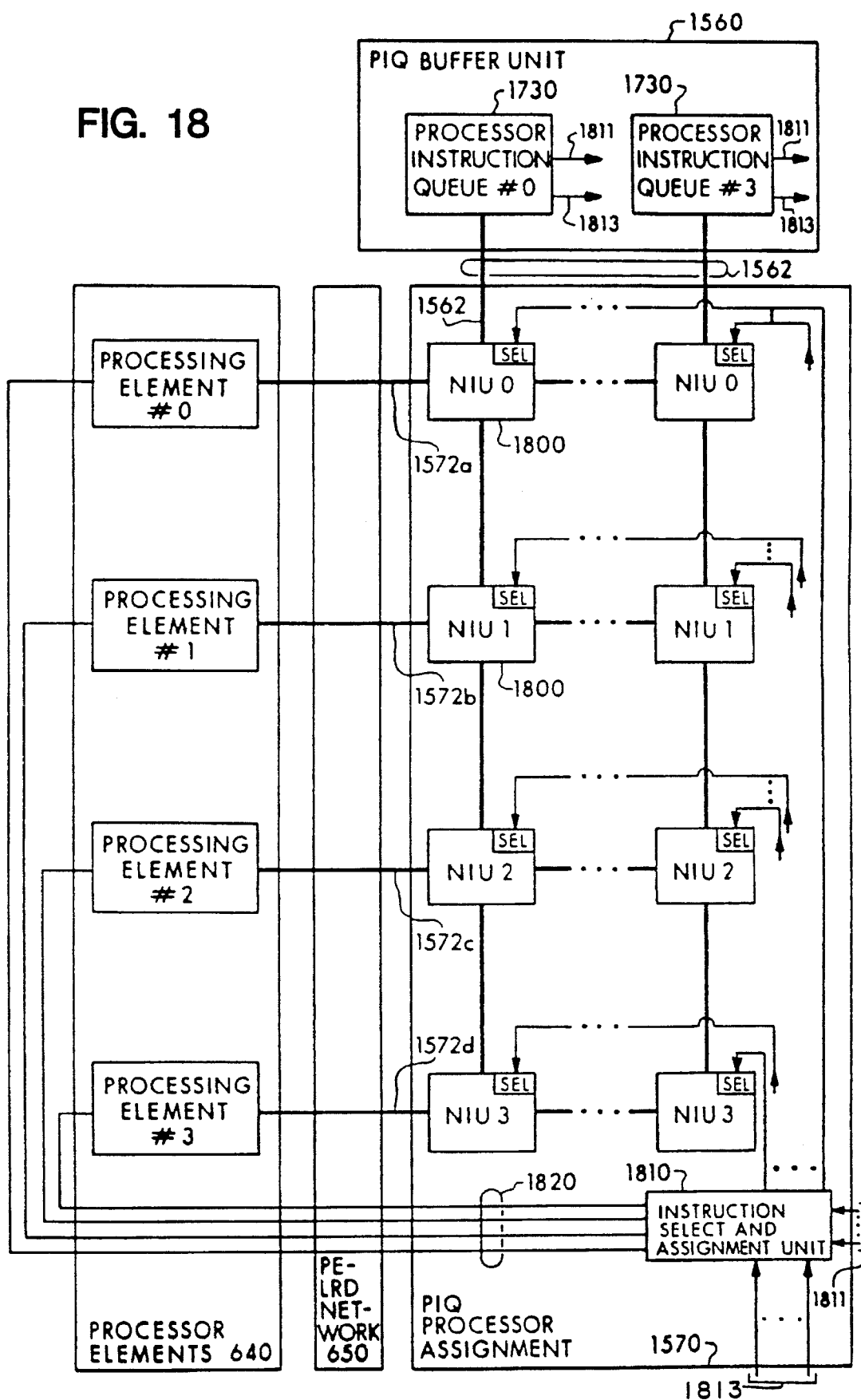*BIAX Corporation v. Intel*
Civil Action No. 2:05-cv-184-TJW

EXHIBIT 3
(PART 2)

FIG. 18

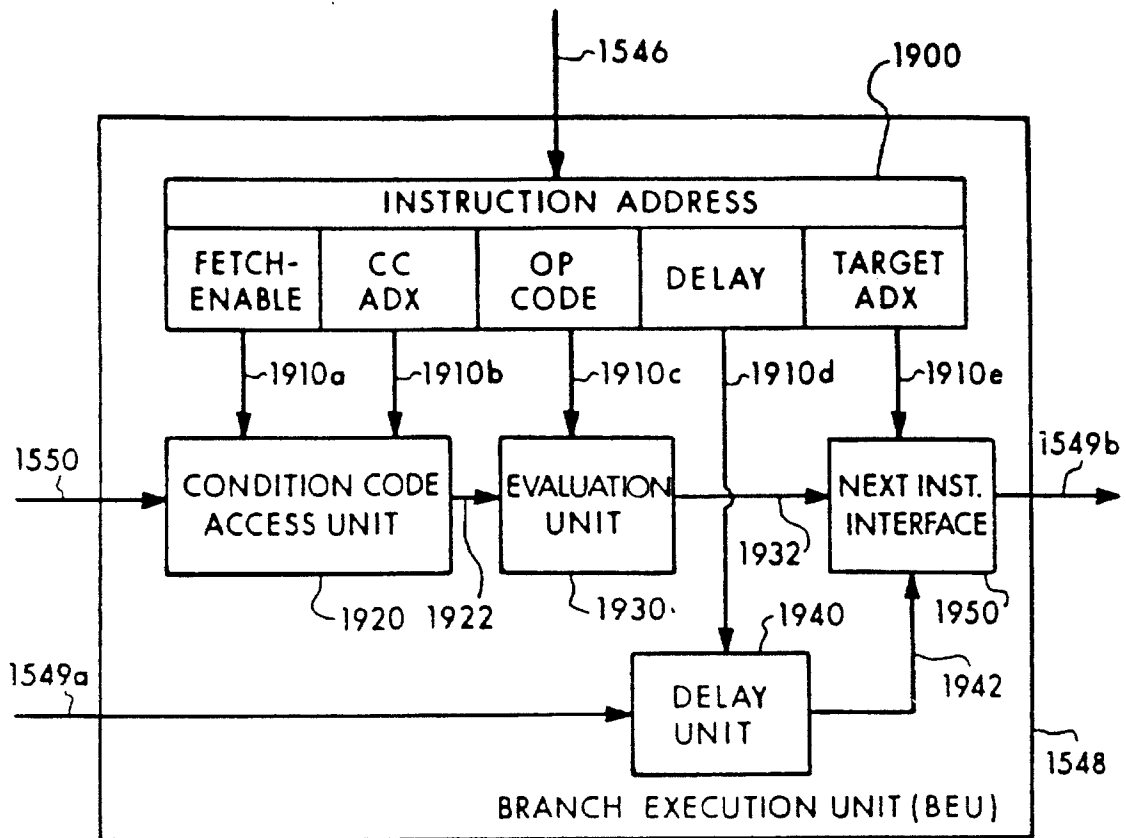**FIG. 19**
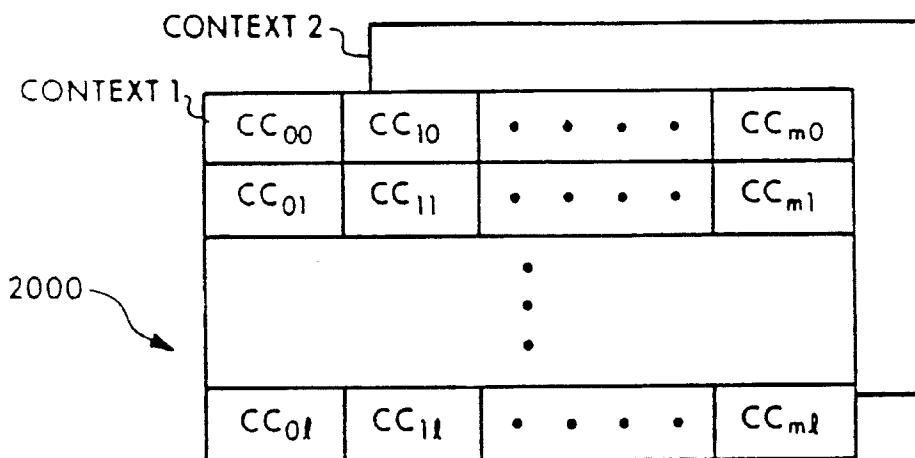

FIG. 20

FIG. 21

TO PIQ
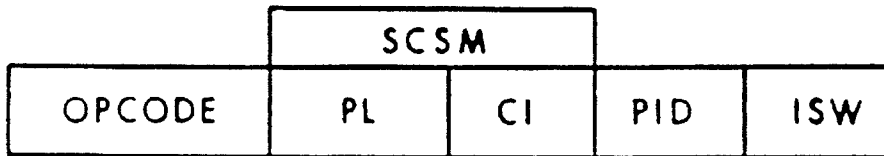~1572

X ___650___

~2102

PROCESSOR ELEMENT

~2100
INST. REG 0

STAGE 1   ~2104 ┌2140

INST. DECODE | REG 0 ┐2146 → | 670 X | TO REG FILE
             | REG 1 ┐2148 →

~2142

2110
INST. REG 1

            ~2114   ┌2150
STAGE 2   ~2144

2112   EFFECTIVE ADDRESS ALU | MEM WRITE ┐2154 → | 670 X | TO REG FILE
                               | CC ┐2156 → | X | TO CC FILE
                                                        670

2120   ~2152
INST. REG 2

        ~2124   ┌2160
STAGE 3

2122   COMPUTE | MEM INTERFACE ┐2164 → | 650 | 1598 | TO DATA CACHE
        CCS | | X

2130   ~2162
INST. REG 3

        ~2132   ┌2170
STAGE 4

        STORE   ┐2172 → | 670 X | REG FILE
        EFF. ADD. UPDATE ┐2174 → | X | REG FILE
        CC STORE   ┐2176 → | X | CC FILE

640

FIG. 22a

	SCSM			
OPCODE	PL	CI	PID	ISW

FIG. 22b

SCSM		
CI	PL	REG ID

FIG. 22c

SCSM		
CI	PL	CC ID

FIG. 22d

ADX	PID	CI	MISC

5,517,628

1

COMPUTER WITH INSTRUCTIONS THAT USE AN ADDRESS FIELD TO SELECT AMONG MULTIPLE CONDITION CODE REGISTERS

This is a continuation of copending application Ser. No. 08/093,794 filed on Jul. 19, 1993, now abandoned, which is a continuation of prior application Ser. No. 07/913,736 filed on Jul. 14, 1992, now abandoned, which is a continuation of prior application Ser. No. 07/560,093 filed on Jul. 30, 1990, now abandoned, which is a division of application Ser. No. 07/372,247 filed on Jun. 26, 1989, now U.S. Pat. No. 5,021,945, which is a division of application Ser. No. 06/794,221, filed on Oct. 31, 1985, now U.S. Pat. No. 4,847,755, issued Jul. 11, 1989.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention generally relates to parallel processor computer systems and, more particularly, to parallel processor computer systems having software for detecting natural concurrencies in instruction streams and having a plurality of identical processor elements for processing the detected natural concurrencies.

2. Description of the Prior Art

Almost all prior art computer systems are of "Von Neumann" construction. In fact, the first four generations of computers are Von Neumann machines which use a single large processor to sequentially process data. In recent years, considerable effort has been directed towards the creation of a fifth generation computer which is not of the Von Neumann type. One characteristic of the so-called fifth generation computer relates to its ability to perform parallel computation through use of a number of processor elements. With the advent of very large scale integration (VLSI) technology, the economic cost of using a number of individual processor elements becomes cost effective.

An excellent survey of multi-processing technology involving the use of parallel architectures is set forth in the June, 1985 Issue of COMPUTER, published by the IEEE Computer Society, 345 East 47th Street, New York, N.Y. 10017, which is not believed to be prior art to the present invention but serves to define the activity in this field. In particular, this issue contains the following articles:

(a) "Essential Issues in Multi-processor Systems" by Gajski et al.;

(b) "Microprocessors: Architecture and Applications" by Patton; and

(c) "Research on Parallel Machine Architecture for Fifth-Generation Computer Systems" by Murakami et al.

In addition, an article in the IEEE Spectrum, November, 1983, entitled "Computer Architecture" by A. L. Davis discusses the features of such fifth-generation machines. Whether or not an actual fifth generation machine has yet been constructed is subject to debate, but various features have been defined and classified.

As Davis recognizes, fifth-generation machines should be capable of using multiple-instruction, multiple-data (MIMD) streams rather than simply being a single instruction, multiple-data (SIMD) system. The present invention is believed to be of the fifth-generation non-Von Neumann type which is capable of using MIMD streams in single context (SC-MIMD) or in multiple context (MC-MIMD). The present invention, however, also finds application in the entire computer classification of single and multiple context

2

SIMD (SC-SIMD and MC-SIMD) machines as well as single and multiple context single-instruction, single data (SC-SISD and MC-SISD) machines.

While the design of fifth-generation computer systems is fully in a state of flux, certain categories of systems have been defined. Davis and Gajski, for example, base the type of computer upon the manner in which "control" or "synchronization" of the system is performed. The control classification includes control-driven, data-driven, and reduction (or demand) driven. The control-driven system utilizes a centralized control such as a program counter or a master processor to control processing by the slave processors. An example of a control-driven machine is the Non-Von-1 machine at Columbia University. In data-driven systems, control of the system results from the actual arrival of data required for processing. An example of a data-driven machine is the University of Manchester dataflow machine developed in England by Ian Watson. Reduction driven systems control processing when the processed activity demands results to occur. An example of a reduction processor is the MAGO reduction machine being developed at the University of North Carolina, Chapel Hill. The characteristics of the non-Von-1 machine, the Manchester machine, and the MAGO reduction machine are carefully discussed in the Davis article. In comparison, data-driven and demand-driven systems are decentralized approaches whereas control-driven systems are centralized. The present invention is more properly categorized in a fourth classification which could be termed "time-driven." Like data-driven and demand-driven systems, the control system of the present invention is decentralized. However, like the control-driven system, the present invention conducts processing when an activity is ready for execution.

It has been recognized by the Patton, Id. at **39**, that most computer systems involving parallel processing concepts have proliferated from a large number of different types of computer architectures. In such cases, the unique nature of the computer architecture mandates or requires either its own processing language or substantial modification of an existing language to be adapted for use. To take advantage of the highly parallel structure of such computer architectures, the programmer is required to have an intimate knowledge of the computer architecture in order to write the necessary software. As a result, porting programs to these machines requires substantial amounts of the users effort, money and time.

Concurrent to this activity, work has also been progressing on the creation of new software and languages, independent of a specific computer architecture, that will expose (in a more direct manner), the inherent parallelism. Hence, most effort in designing supercomputers has been concentrated at the hardware end of the spectrum with some effort at the software end.

Davis has speculated that the best approach to the design of a fifth-generation machine is to concentrate efforts on the mapping of the concurrent program tasks in the software onto the physical hardware resources of the computer architecture. Davis terms this approach one of "task-allocation" and tauts it as being the ultimate key to successful fifth-generation architectures. He categorizes the allocation strategies into two generic types. "Static allocations" are performed once, prior to execution, whereas "dynamic allocations" are performed by hardware whenever the program is executed or run. The present invention utilizes a static allocation strategy and provides task allocations for a given program after compilation and prior to execution. The recognition of the "task allocation" approach in the design of

5,517,628

3

fifth generation machines was used by Davis in the design of his "Data-driven Machine-II" constructed at the University of Utah. In the Data-driven Machine-II, the program was compiled into a program graph that resembles the actual machine graph or architecture.

Task allocation is also referred to as "scheduling" in the Gajski article. Gajski sets forth levels of scheduling to include high level, intermediate level, and low level scheduling. The present invention is one of low-level scheduling, but it does not use conventional scheduling policies of "first-in-first-out", "round-robin", "shortest type in job-first", or "shortest-remaining-time." Gajski also recognizes the advantage of static scheduling in that overhead costs are paid at compile time. However, Gajski's recognized disadvantage, with respect to static scheduling, of possible inefficiencies in guessing the run time profile of each task is not found in the present invention. Therefore, the conventional approaches to low-level static scheduling found in the Occam language and the Bulldog compiler are not found in the software portion of the present invention. Indeed, the low-level static scheduling of the present invention provides the same type, if not better, utilization of the processors commonly seen in dynamic scheduling by the machine at run time. Furthermore, the low-level static scheduling of the present invention is performed automatically without intervention of programmers as required (for example) in the Occam language.

Davis further recognizes that communication is a critical feature in concurrent processing in that the actual physical topology of the system significantly influences the overall performance of the system.

For example, the fundamental problem found in most data-flow machines is the large amount of communication overhead in moving data between the processors. When data is moved over a bus, significant overhead, and possible degradation of the system, can result if data must contend for access to the bus. For example, the Arvind data-flow machine, referenced in Davis, utilizes an I-structure stream in order to allow the data to remain in one place which then becomes accessible by all processors. The present invention teaches a method of hardware and software based upon totally coupling the hardware resources thereby significantly simplifying the communication problems inherent in systems that perform multiprocessing.

Another feature of non-Von Neumann type multiprocessor systems is the level of granularity of the parallelism being processed. Gajski terms this "partitioning." The goal in designing a system according to Gajski is to obtain as much parallelism as possible with the lowest amount of overhead. The present invention performs concurrent processing at the lowest level available, the "per instruction" level. The present invention teaches a method whereby this level of parallelism is obtainable without execution time overhead.

Despite all of the work that has been done with multiprocessor parallel machines, Davis (Id. at 99) recognizes that such software and/or hardware approaches are primarily designed for individual tasks and are not universally suitable for all types of tasks or programs as has been the hallmark with Von Neumann architectures. The present invention sets forth a computer system that is generally suitable for many different types of tasks since it operates on the natural concurrencies existent in the instruction stream at a very fine level of granularity.

The patent invention, therefore, pertains to a non-Von Neuman MIMD computer system capable of operating upon many different and conventional programs by a number of

4

different users. The natural concurrencies in each program are statically allocated, at a very fine level of granularity, and intelligence is added to each instruction at essentially the object code level. The added intelligence includes a logical processor number and an instruction firing time in order to provide the time-driven decentralized control for the present invention. The detection and low level scheduling of the natural concurrencies and the adding of the intelligence occurs only once for a given program, after conventional compiling of the program, without user intervention and prior to execution. The results of this static allocation are executed on a system containing a plurality of identical processor elements. These processor elements are characterized by the fact that they contain no execution state information from the execution of previous instructions, i.e., they are context free. In addition, a plurality of contexts, one for each user, are provided wherein the plurality of context free processor elements can access any storage resource contained in any context through total coupling of the processor element to the shared resource during the processing of an instruction. Under the teachings of the present invention no condition code or results registers are found on the individual processor elements.

Based upon the features of the present invention, a patentability investigation was conducted resulting in the following references:

ARTICLES

Dennis, "Data Flow Supercomputers", Computer, November, 1980, Pgs. 48–56.

Hagiwara, H. et al., "A Dynamically Microprogrammable, Local Host Computer With Low-Level Parallelism", IEEE Transactions on Computers, C-29, No. 7, July, 1980, Pgs. 577–594.

Fisher et al., "Microcode Compaction: Looking Backward and Looking Forward", National Computer Conference, 1981, Pgs. 95–102.

Fisher et al., "Using an Oracle to Measure Potential Parallelism in Single Instruction Stream Programs", IEEE No. 0194-1895/81/0000/0171, 14th Annual Microprogramming Workshop, Sigmicro, October, 1981, Pgs. 171–182.

J. R. Vanaken et al., "The Expression Processor, IEEE Transactions on Computers, C-30, No. 8, August, 1981, Pgs. 525–536.

Bernhard, "Computing at the Speed Limit", IEEE Spectrum, July, 1982, Pgs. 26–31.

Davis, "Computer Architecture", IEEE Spectrum, November, 1983, Pgs. 94–99.

Hagiwara, H. et al., "A User-Microprogrammable Local Host Computer With Low-Level Parallelism", Article, Association for computing Machinery, #0149-7111/83/0000/0151, 1983, Pgs. 151–157.

McDowell, Charles Edward, "SIMAC: A Multiple ALU Computer", Dissertation Thesis, University of California, San Diego, 1983(111 pages).

McDowell, Charles E., "A Simple Architecture for Low Level Parallelism", Proceedings of 1983 International Conference on Parallel Processing, Pgs. 472–477.

Requa, et al., "The Piecewise Data Flow Architecture: Architectual Concepts, IEEE Transactions on Computers, Vol. C-32, No. 5, May, 1983, Pgs. 425–438.

Fisher, A. T., "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", Computer, 1984 Pgs 45–52.

5,517,628

<table>
<tr><td>

5

Fisher, et al., "Measuring the Parallelism Available for Very Long Instruction, Word Architectures", IEEE Transactions on Computers, Vol. C-33, No. 11, November, 1984, Pgs. 968–976.

PATENTS

Freiman, et al., (U.S. Pat. No. 3,343,135), "Compiling Circuitry for a Highly-Parallel Computing System", Sep. 19, 1967.

Reigel, (U.S. Pat. No. 3,611,306), "Mechanism to Control the Sequencing of Partially Ordered Instructions in a Parallel Data Processing System", Oct. 5, 1971.

Culler, (U.S. Pat. No. 3,771,141), "Data Processor With Parallel Operations for Instructions", Nov. 6, 1973.

Gruner, (U.S. Pat. No. 4,104,720), "CPU/Parallel Processor Interface with Microcode Extension", Aug. 1, 1978.

Blum, et al. (U.S. Pat. No. 4,109,311), "Instruction Execution Modification Mechanism for Time Slice Controlled Data Processors", Aug. 22, 1978.

Dennis, et al. (U.S. Pat. No. 4,153,932), "Data Processing Apparatus for Highly Parallel Execution of Stored Programs", May 8, 1979.

Kober, (U.S. Pat. No. 4,181,936), "Data Exchange Processor for Distributed Computing System", Jan. 1, 1980.

Bernhard, (U.S. Pat. No. 4,228,495), "Multiprocessor Numerical Control System", Oct. 14, 1980.

Gilliland, et al., (U.S. Pat. No. 4,229,790), "Concurrent Task and Instruction Processor and Method", Oct. 21, 1980.

Caril, (U.S. Pat. No. 4,241,398), "Computer Network Line Protocol System", Dec. 23, 1980.

Koehler, et al., (U.S. Pat. No. 4,270,167), "Apparatus and Method for Cooperative and Concurrent Coprocessing of Digital Information", May 26, 1981.

Lorie, et al., (U.S. Pat. No. 4,435,758), "Method for Conditional Branch Execution in SIMD Vector Processors", Mar. 6, 1984.

DeSantis, (U.S. Pat. No. 4,468,736), "Mechanism for Creating Dependency Free Code For Multiple Processing Elements", Aug. 28, 1984.

DeSantis, (U.S. Pat. No. 4,466,061), "Concurrent Processing Elements For Using Dependency Free Code", Aug. 14, 1984.

In the two McDowell references, a parallel processing system utilizing low level parallelism was disclosed. McDowell differentiates between low level and high level parallelism in that low level parallelism pertains to the execution of two or more machine level operations in parallel whereas high level parallelism is the parallel execution of high level language constructs which includes source language statements, tasks, procedures, or entire programs.

Like the present invention, McDowell identifies and statically schedules low level parallel operations found in existing "sequential" programming languages. Further, he does not require the creation of special purpose programming languages or user modifications to the parallelism existing with the program at compile time, rather he simply analyzes the low level parallelism existing within the basic blocks (BBs) that make up the program.

In the McDowell SIMAC system, processor elements are identical and communicate through a shared memory and a set of shared registers. The processor elements are controlled by a control processor that is fed by a single instruction stream.

</td><td>

6

McDowell recognizes his particular parallel processing system does not fit into the standard categories of: homogeneous multi-processors, non-homogeneous multi-processors, array processors, and pipeline processors. Rather, McDowell classified his computer system in the class of Schedulable Parallel Instruction Execution (SPIE) computers. McDowell's reasoning as to why SIMAC is categorized as a SPIE processor is as follows:

> SPIE processors have only one instruction stream and are therefore immediately eliminated from the first two groups. They have some of the characteristics of array processors, but they are distinctly different in that each of the several processing elements may be executing different operations. In array processors all processing elements are restricted to executing the same instruction on different data. SPIE processors are also similar in capability to pipelined machines but again they are distinct in that each processing element may be given a new operation each instruction cycle that is independent of the operation it was performing in the previous cycle. Pipelined machines only specify a single operation per instruction cycle, and then other operations on other processing elements may be triggered from that one initial operation. Dissertation Thesis at 11.

The present invention also is properly categorized as a SPIE processing system since each processor element may be given a new operation each instruction cycle that is independent of the operation it was performing in the previous cycle. Hence, the McDowell reference provides important background material to the teachings of the present invention. However, important distinctions exist between McDowell's SIMAC system and the system of the present invention. SIMAC is quoted as being a SPIE machine that can perform scalar concurrent execution of a single user's program (i.e., SIMD). The present invention also fits into this SIMD category, however, it is further capable of performing scalar concurrent execution for multiple instruction streams (i.e., MIMD) and further for single or multiple context (i.e., SC-MIMD or MC-MIMD).

SIMAC implicitly assigns instructions (termed "machine operations") to processor elements by virtue of the physical ordering of the instructions when they are grouped together into "parallel instructions." The identity of the assigned processor is not a part of the instruction per se. The present invention teaches that the order of the instructions within a group of concurrently executable instructions does not determine the processor assignment. Rather, the processor assignment, under the teachings of the present invention, is explicitly made by adding a logical processor number to become physically part of the instruction which is performed in software prior to execution. In addition, the present invention distinguishes the logical processor number from the actual physical processor number that executes the instruction. This allows for further machine independence of the software in that the physical processor assignment is performed dynamically at execution time.

SIMAC code generation can be divided into three components during which the static allocation and scheduling of its instructions are performed (prior to execution). First, is the determination of the machine operations to be performed. This is the typical code generation phase of a compiler. Second, is the formation of these operations into parallel instructions containing the operations that can be done in parallel. The third component comprises the ordering (scheduling) of the parallel instructions so

> ". . . to allow the execution of two [parallel instructions] containing different length [machine operations] to

</td></tr>
</table>

5,517,628

7

overlap using only simple hardware controls . . . "
McDowell's Conference Paper @ Pg. 475.

In other words, McDowell attempts to schedule instructions with differing length execution times so that they can execute concurrently with a minimum amount of hardware support. To the contrary, the present invention does not require this third component. The use of the logical processor number and the instruction firing time that are physically attached to each instruction preclude the need for this phase.

As stated above, SIMAC performs its concurrency detection during compile time only. As a result, the static analysis of the concurrency present within the instruction stream is language dependent. In contrast, the concurrency detection of present invention is language independent since the detection takes place after compilation and prior to execution. The present invention can perform its concurrency detection and scheduling without requiring any modifications to the compiler.

Like SIMAC, the present invention is designed to be independent of the number of processor elements contained within the system. The processor elements in SIMAC, however, are conventional load/store RISC style processors that contain contextual information regarding the state of the previous execution status. These are the "T" and "V" bits associated with each of the SIMAC processor elements. These are used for branching and are the result of executing certain comparison and test instructions. The processor elements of the present invention are context free processors that support a RISC-like instruction set. The term "context free" means that the processor elements contain no state information, e.g., condition codes, registers, program status words, flags, etc. Like SIMAC, the processor elements of the present invention contain no local registers. All operations are performed on data already stored in an array of registers and are accessible by any of the individual processor elements through a register switch.

In SIMAC, only a single set (or level) of 32 registers is available and is shared by all processor elements. Each register may be physically read by any or all processor elements simultaneously. However, only one processor element may write into a specific register at any one time. (Dissertation Thesis at 26) Whereas the registers of this invention also have this characteristic, the architecture of the register set in the present invention is much different than that of the SIMAC machine. The SIMAC machine contains a single level of registers. The present invention contains a number of levels of register sets; each level corresponding to a procedural depth relative to the main program. In addition, each set of levels is replicated to support each context or user that is currently being executed. Thus, under the teachings of the present invention, contexts may be switched or procedures entered without having to flush any registers to memory.

SIMAC does not discuss the need for the production and assignment of condition codes. Without the use of multiple condition code sets, it is not possible to distribute instructions which affect the condition codes across multiple processor elements and guarantee proper condition code data for the execution of subsequent dependent instructions. The present invention teaches a method for the management and assignment of multiple condition code sets. The management and assignment of this resource is analogous to that done for the register resources.

SIMAC also requires a separate control processor that performs the branching operations. This processor contains the program counter and executes the branch operations Each processor element in SIMAC contains three status bits

8

used in the branching operations. The present invention does not use a control processor, nor does it use a program counter and its processor elements contain no such status bits. SIMAC is a centralized control system under the Davis classification whereas the present invention is decentralized.

Hence, the present invention is significantly more general than the SIMAC approach since the present invention performs inter-program parallelism as well as intra-program parallelism.

In addition, while SIMAC provides a register array, McDowell indicates that this, perhaps is not desirable because of cost and proposes that future work be done on an arrangement other than shared registers (Id. at 79) The present invention not only makes use of the shared register concept but also provides sharing of condition codes and the management of those condition codes as well. While McDowell, implicitly, assigns the machine operations to his processor elements, the present invention explicitly extends each instruction with a logical processor number as well as providing a specific firing time for the instruction. There is no disclosure in SIMAC of either a firing time or a logical processor number. Therefore, the master controller, in SIMAC, uses a program counter that provides the control for the processor elements. Under the teaching of the present invention, a set of logical resource drivers (one for each context or user) is provided and instruction selection is based upon the firing times added to each instruction within a basic block (i.e., time driven).

DeSantis sets forth in U.S. Pat. Nos. 4,466,061 and 4,468,736 a concurrent data processor which is:

adapted to receive strings of object code, form them into higher level tasks and to determine sequences of such tasks which are logically independent so that they may be separately executed ('736 patent at col. 2, lines 50–54).

The logically independent sequences are separately executed by a plurality of processor elements. The first step in the DeSantis approach is to hardware compile the program into strings of object code or machine language in a form which is particularly designed for the computer architecture to provide a dependency free code. The DeSantis invention then forms an independency queue so that all processing for the entire queue is accomplished in one step or cycle. The hardware structure for the DeSantis approach is set forth in FIG. 2 of the patent and contemplates the use of a number of small processor elements (SPEs) each with local memories.

The nature of the interconnection between the local memories of each SPE and the direct storage module 14 is not clear. The patent states:

In the meantime, respective data items required for execution have been fetched from main memory and stored at the appropriate locations in local memories which locations are accessed by the pointers and the job queue ('736 patent at Col. 6, Lines 19–23).

It does not appear from this description, that each individual SPE has access to the local memories of the other SPE rather, this allocation seems to be made by the direct storage module (DSM). Hence, under DeSantis, the queue which contains the independent sequences for processing delivers the string to an identified processor element, so that the processor elements can process all strings concurrently and in parallel, the direct storage module must deliver the necessary results into the local memories for each SPE. With respect to each individual SPE, DeSantis states:

Since the respective processors are provided to execute different functions, they can also be special purpose

5,517,628

9

microprocessors containing only that amount of logic circuitry required to perform the respective functions. The respective circuits are the arithmetic logic unit, shift unit, multiply unit, indexing unit, string processor and decode unit ('736 patent at Col. 7, Lines 7–13).

The primary difference between the approach of the present invention and DeSantis relates to when the detection of the concurrency occurs. TDA performs its concurrency detection during a pre-processing stage at the object level using information normally thrown away by the source code level language processors. DeSantis performs it with part of the hardware collectively called the cache mechanism **10** (see FIG. 1) dynamically during execution time. Hence, the DeSantis approach constantly uses overhead and resources to hardware de-compile each program. A secondary difference relates to the execution of individual streams of dependent instructions. DeSantis requires that a stream of dependent instructions be executed on the same processor. To the contrary, the present invention permits the execution of a stream of dependent instructions on the same or multiple processor elements.

It is believed that the aforesaid McDowell and DeSantis references are the most pertinent of the references discovered in the patentability investigation and, therefore, more discussion occurs for it than the following references.

J. R. Vanaken in "The Expression Processor" article classifies his processor as a "direct descendant of the tree processor." There is a similarity between the Vanaken architecture and the architecture of the present invention wherein each processor element is capable of accessing, through an alignment network, any one of a number of registers located in a register array. Vanaken utilizes thirty-two processor elements and eight register modules in the register storage. A crossbar switch is disclosed for the alignment network. In this configuration, the identical processor elements of Vanaken do not exchange data directly with the main memory. Rather, data transfers take place between the register storage and the memory. This is not the configuration of the present invention. Vanaken further contemplates utilizing a separate compiler (only for scaler programs) for the detection of low level parallelism and for the assignment of detected parallelism to individual processor elements. How this is done, however, is not disclosed. Vanaken simply discloses the desired goals:

> First, it [the compiler] must detect the potential parallelism in the program. Second, it must map detected parallelism onto the structure of the target machine. *The Expression Processor* at 535.

Vanaken refers only to the parallelism detection techniques presented by Kuck. However, Vanaken's tree structured processor element architecture requires that the computational task be modeled as a computational tree, or as a computational wavefront (wavefront ref: Kuck's work). In the former case, processor assignment is accomplished by assigning a processor to each node in the tree. If there are more nodes in the tree than processors, the task must be broken down into smaller subtasks that will fit on the processor array. In the latter case, the wavefront propagates from the lowest level of processors to the next higher level and so on until the highest level is achieved. Again, if the number of processors required by the wavefront is greater than the number of processors available, the task must be broken into smaller subtasks. This type of flow through ordered processors does not exist in the present invention.

In the Hagiwara articles, the disclosed MIMD QA-1 computer system appears to be a very long instruction word (VLIW) machine having a 160 bit or 80 bit word. The

10

Hagiwara references term this a "horizontal-type microinstruction" and the QA-1 machine takes advantage of the low-level parallel concurrency, at the microcode level. The QA-2 system has 62 working registers constructed of large capacity/high-speed RAM chips interconnected over a network to each of the four processor elements. The earlier QA-1 system utilized a stack of only 15 working registers. Hence, the QA-2 hardware design contemplates the use of individual processor elements having full access to a number of working registers through a switching network. In QA-2, through use of the network, the results of one processor element are delivered to a working register which is then delivered to another processor element as input data. Hence, the QA-1 and QA-2 configurations take advantage of scaler concurrency, multiple ALUs, and the sharing of working registers. The QA-2 approach modifies the QA-1 approach by providing special purpose registers in the register file (i.e., constant, general, indirect, and special); such a division, however, is not apparent in the QA-1 architecture. From a compiling point of view, the QA-1 was primarily designed to be encoded by a programmer at the microcode level to take programming advantage of the inherent power of the system's architecture. The QA-2 machine which is directed towards the personal computer or local host computer (e.g., for laboratory use) contemplates that programs written in high-level languages will be compiled into the QA-2 microprogrammed interpretations.

The four Fisher references disclose a SIMD system based upon a "very long instruction word" (VLIW). The goal presented in these articles is to determine the low level parallelism or "fine-grained parallelism" of a program in a separate compiler so that all individual movements of the data within the VLIW machine are completely specified at compile time. At the outset, it is important to recognize that the hardware architecture for implementing Fisher's VLIW machine is not discussed in these references other than in a general sense. Fisher contemplates:

> Each of the eight processors contain several functional units, all capable of initiating an operation in each cycle. Our best guess is two integer ALUs, one pipelined floating ALU, one memory port, several register banks, and a limited crossbar for all of these to talk to each other. *Computer* article at pg. 50.

Rather, the Fisher articles concentrate on the software compiling technique called "trace scheduling" and nicknamed BULLDOG. Fisher further requires an instruction word of "fixed length." The length of Fisher's instruction word, therefore, dictates the amount of hardware requi d to process the instruction word. In that respect, the   ent invention is much more dynamic and fluid since it can have any number of processors wherein each processor processes the parallel instruction during the instruction firing time.

Lorie et al. (U.S. Pat. No. 4,435,758) sets forth a technique for ordering instructions for parallel execution during compile time by adding code to the instruction stream. The present invention does not add code to the instruction stream.

The Reigel patent (U.S. Pat. No. 3,611,306) relates to the ordering of the detected parallel instructions for a particular computer architectural approach.

The patent issued to Freiman, et al. (U.S. Pat. No. 5,343,135) teaches a dynamic compiling system, hardware based, for use in a multiprocessor environment for analyzing particular types of mathematical expressions for parallel execution opportunities and then to automatically assign individual operations to processors within the system. Freiman first identifies the concurrencies, in the hardware, and

5,517,628

11

determines the time periods within which the operations may be performed. These determined times are "the earliest times in which a given operation may be performed." The Freiman invention also analyzes the particular mathematical expression and determines, in the hardware, "the latest times in which an operation can be performed." This information generated from the analysis of the mathematical algorithm is stored in a word register which is then used in the multi-processor environment. Each processor has associated with it a processor control which for all processors is identical and which is driven by six control fields. One function of the control block is to determine whether or not a particular processor is ready to receive a job, to indicate when it has received a job, and to analyze each job to see if the processor can proceed immediately with the job or whether it must wait until one of the operants of the job is already assigned. Once a processor is assigned to a task the necessary data contained in the results register may not be available at that particular time and hence the processor must wait until a test of the results register indicates that the result is available. Once the result is available, the processor will perform its operation and place its result in the appropriate result register for access by another processor. All processors in Freiman have access to the results register by means of a crossbar switch.

The patent issued to Dennis et al. (U.S. Pat. No. 4,153, 932) teaches a highly parallel multiprocessor for the concurrent processing of programs represented in data flow form. Specifically, a mathematical computation as shown at column 5, lines 17–23 and as represented in FIG. **2** in data flow language is used as an example. Hence, the data flow form is generated and stored in the memory of the processor in designated instruction cells wherein each instruction cell corresponds to an operator in the data flow program. In operation, when an instruction cell is complete (i.e., containing the instruction and all necessary operands), a signal is generated to the Arbitration Network which directs the flow of the information in the instruction cell to the identified processor. The processor operates on the information and delivers it back through a distribution network into the main memory.

The patent issued to Culler (U.S. Pat. No. 3,771,141) discloses a high speed processor capable of parallel operations on data. Multiple or parallel processors, however, are not used. Rather, the parallelism is achieved structurally by implementing each of the processor's data registers with four separate multi-bit data input ports. Hence, four operations can be performed at once. This results in a "tightly coupled" arrangement since the data from any given processor register is ready to be received by any other register at the next clock cycle. The control of which register is to receive which information occurs at the object code level and, as shown in Table 1 of Culler, four additional fields are provided in each object code instruction for controlling the flow of data among the processor's registers. Culler states:

> The data pad memory is tightly coupled to the arithmetic unit and serves as an effective buffer between the high speed arithmetic unit and a large capacity random access core memory. As an indication of the effective speed, a complete multiplication of two signed eight bit words can be accomplished in three cycles or 375 nanoseconds.

The Gilliland et al. patent (U.S. Pat. No. 4,229,790) sets forth a processor for processing different and independent jobs concurrently through the use of pipelining with precedent constraints. The Gilliland invention is capable of processing programming tasks concurrently as well as process-

12

ing the parallel parts of each programming task concurrently. It appears that the invention operates on concurrencies in the program at the source code level rather than at the object code level. The Gilliland processor is capable of processing up to 128 parallel processor paths.

The patent issued to Blum et al. (U.S. Pat. No. 4,109,311) relates to a data processing system based upon time slice multiprogramming. The Blum invention contemplates a conventional multiprocessor arrangement wherein a first processor controls a keyboard and display interface, a second processor controls printer, disk storage, and tape storage interface, and a third processor executes the problem oriented programs located in the main storage.

Gruner (U.S. Pat. No. 4,104,720) sets forth a CPU control multiprocessor system wherein each parallel processor is directly interfaced through an interface control circuit to the CPU. Each interface circuit contains memory for storing portions of the micro instructions contained within the CPU. Gruner calls this an "extension" of the micro instructions from the CPU into the interface circuit for each parallel processor. The Gruner system while controlling each interface circuit to allocate and to assign concurrent tasks utilizes a conventional bus structure for the transfer of information between the processors.

The patent issued to Caril (U.S. Pat. No. 4,241,398) relates to a supervisory control system wherein a central processing unit controls and supervises the operation of a number of remote processing units. The Caril invention relates to a low overhead line protocol for controlling the asynchronous exchange of communication information between the central processing unit and each remote processing unit.

Bernhard et al. (U.S. Pat. No. 4,228,495) relates to multiprocessor numerical control system. The Bernhard invention relates to a main processor coupled over a bus structure to a plurality of programmable interface processors.

Koehler et al. patent (U.S. Pat. No. 4,270,167) discloses a multiprocessor arrangement wherein a central processor has primary control and access to a local bus and wherein the remaining plurality of processors also share access to the local bus. Arbitration among the processors is provided to the local bus as well as the generation of query status and processor status signals.

The patent issued to Kober (U.S. Pat. No. 4,181,936) relates to a distributed computing system for increasing the efficiency of the data bus.

The Requa article discusses a piecewise data flow architecture which seeks to combine the strengths of other supercomputers. The goal of the Requa architecture is to take the source code program and recompile it into "basic blocks." These basic blocks are designed to be no longer than 255 instructions and the natural concurrencies existing in each basic block are identified and processed separately by a number of "scaler processors." The natural concurrencies are encoded as part of the instructions to the scaler processors. The concurrent instructions waiting for execution reside in a stack of registers to which all of the scaler processors has access to deliver data into and to receive data from. As in the Dennis approach, when a particular instruction residing in the register is completed (i.e., when a prior data dependency has been satisfied and stored) the scaler instruction apparently raises a flag which will be detected for delivery to a scaler processor.

In the Bernhard article, the author surveys the current state of art concerning supercomputer design. The article discusses the current supercomputer research projects and

5,517,628

13

provides critical comments concerning each type of design. The disclosure in this article appears to be cumulative to the analysis set forth above and none of the approaches set forth suggest the TDA processor register communication technique.

SUMMARY OF INVENTION

The present invention provides a method and a system that is non-Von Neuman and one which is adaptable for use in single or multiple context SISD, SIMD, and MIMD configurations. The method and system is further operative upon a myriad of conventional programs without user intervention.

The present invention statically determines at a very fine level of granularity, the natural concurrencies in the basic blocks (BBs) of programs at essentially the object code level and adds a logical processor number (LPN) and an instruction firing time (IFT) to each instruction in each basic block in order to provide a time driven decentralized control. The detection and low level scheduling of the natural concurrencies and the adding of the intelligence occurs only once for a given program after conventional compiling and prior to execution. At this time all instruction resources are assigned.

The present invention further executes the basic blocks containing the added intelligence on a system containing a plurality of identical processor elements each of which does not retain execution state information from prior operations. Hence, all processor elements are context free. Instructions are selected for execution based on instruction firing time. Each processor element is capable of executing instructions on a per-instruction basis such that dependent instructions can execute on the same or different processor elements. A given processor element in the present invention is capable of executing an instruction from one context followed by an instruction from another context. All context information necessary for processing a given instruction is contained elsewhere in the system.

The system and method of the present invention are described in the following drawing and specification.

DESCRIPTION OF THE DRAWING

FIG. 1 is the generalized flow of the TOLL software of the present invention;

FIG. 2 is a graphic representation of a sequential series of basic blocks found within the conventional compiler output;

FIG. 3 is a graphical presentation of the extended intelligence added to each basic block under the teachings of the present invention;

FIG. 4 is a graphical representation showing the details of the extended intelligence added to each instruction within a given basic block;

FIG. 5 is the breakdown of the basic blocks carrying the extended information into discrete execution sets;

FIG. 6 is a block diagram presentation of the architectural structure of the present invention;

FIGS. 7a–7c represents an illustration of the network interconnections during three successive instruction firing times;

FIGS. 8–11 are the flow diagrams setting forth the program features of the software of the present invention;

FIG. 12 is the flow diagram for determining the execution sets in the TOLL Software;

14

FIG. 13 sets forth the register file organization of the present invention;

FIG. 14 illustrates the transfers between registers in levels during a subroutine call;

FIG. 15 sets forth the structure of the logical resource drivers (LRDs) of the present invention;

FIG. 16 sets forth the structure of the instruction caches control and of the caches of the present invention;

FIG. 17 sets forth the structure of the PIQ buffer unit and the PIQ bus interface unit of the present invention;

FIG. 18 sets forth interconnection of the processor elements through the PE-LRD network to the PIQ processor alignment circuit of the present invention;

FIG. 19 sets forth the structure of the branch execution unit of the present invention;

FIG. 20 illustrates the organization of the contexts of the present invention;

FIG. 21 sets forth the structure of one embodiment of the processor element of the present invention; and

FIGS. 22(a) through 22(d) sets forth the data structures in the processor element of FIG. 21.

GENERAL DESCRIPTION

1. Introduction

In the following two sections, a general description of the software and hardware of the present invention takes place. The system of the present invention is designed based upon a unique relationship between the hardware and software components. While many prior art approaches have primarily provided for multiprocessor parallel processing based upon a new architecture design or upon unique software algorithms, the present invention is based upon a unique hardware/software relationship. The software of the present invention provides the intelligent information for the routing and synchronization of the instruction streams through the hardware. In the performance of these tasks, the software spatially and temporally manages all user accessible resources, e.g., registers, condition codes, memory and stack pointers. This routing and synchronization is done without any user intervention, and does not require changes to the source code. Additionally, the analysis of an instruction stream to provide the additional intelligent information for routing and synchronization is performed only once during the program preparation process (i.e., static allocation) on a given piece of software, and is not performed during execution (i.e, dynamic allocation) as is found in some conventional prior art approaches. The analysis is performed on the object code output from conventional compilers and therefore is programming language independent.

2. General Software Description

In FIG. 1, the general description of the software of the present invention is set forth and is generally termed "TOLL." The software TOLL located in a processing system **160** operates on standard compiler output **100** which is typically object code or an intermediate object code such as "p-code." It is known that the output of conventional compilers is a sequential stream of object code instructions hereinafter referred to as the instruction stream. Conventional language processors typically perform the following functions in generating the sequential instruction stream:

1. lexical scan of the input text,

2. syntactical scan of the condensed input text including symbol table construction,

3. performance of machine independent optimization including parallelism detection and vectorization, and

5,517,628

15

4. an intermediate (PSEUDO) code generation including instruction functionality, resources required, and structural properties.

In the creation of the sequential instruction stream, the conventional compiler creates a series of basic blocks (BBs) which are single entry single exit (SESE) groups of contiguous instructions. See, for example, *Principles of Compiler Design*, Alfred v. Aho and Jeffery D. Ullman, Addison Wesley, 1979, pg. 6, 409, 412–413 and *Compiler Construction for Digital Computers*, David Gries, Wiley, 1971. The conventional compiler, although it utilizes basic block information in the performance of its tasks, provides an output stream of sequential instructions without any basic block designations. The TOLL software of the present invention is designed to operate on the formed basic blocks (BBs) which are created within a conventional compiler. In each of the conventional SESE basic blocks there is exactly one branch (at the end of the block) and there are no control dependencies; the only relevant dependencies within the block are those between the resources required by the instructions.

The output of the compiler **100** in the basic block format, is shown in FIG. 2. The TOLL software **110** of the present invention being processed in a computer **160** performs three basic determining functions on the compiler output **100** as shown in FIG. 1. These functions are to analyze the resource usage of the instructions **120**, extend intelligence for each instruction in each basic block **130**, and to build execution sets composed of one or more basic blocks **140**. The resulting output of these three basic functions **120**, **130**, and **140** from processor **160** comprise the TOLL output **150** of the present invention.

At the outset, the TOLL software of the present invention operates on a compiler output **100** only once and without user intervention. Therefore, for any given program, the TOLL software need operate on the compiler output **100** only once.

The functions of the TOLL software **110** are to analyze the instruction stream in each basic block for natural concurrencies, to perform a translation of the instruction stream onto the actual hardware system of the present invention, to alleviate any hardware induced idiosyncracies that may result from this translation and to encode the resulting stream into an actual machine language to be used on the hardware of the present invention. The TOLL software **110** performs these features by analyzing the instruction stream and then assigning context free processor elements and resources as a result thereof. The TOLL software **110** provides the "synchronization" of the overall system by assigning appropriate firing times to each instruction in the instruction stream.

Instructions can be dependent on one another in a variety of ways although there are only three basic types of dependencies. First, there are procedural dependencies due to the actual structure of the instruction stream; i.e., instructions may follow one another in other than a sequential order due to branches, jumps, etc. Second, operational dependencies are due to the finite number of hardware elements present in the system. These hardware elements include the registers, condition codes, stack pointers, processor elements, and memory. Thus if two instructions are to execute in parallel, they must not require the same hardware element unless they are both reading that element (provided of course, that the element is capable of being read simultaneously). Finally, there are data dependencies between instructions in the instruction stream. This form of dependency will be discussed at length later. However, within a basic block only data and operational dependencies are present.

16

The TOLL software **110** maintains the proper execution of a program; i.e., TOLL must assure that the parallelized code **150** generates the same results as those of the original serial code. In order to do this, the parallelized code **150** must access the resources in the same relative sequence as the serial code for instructions that are dependent on one another; i.e., the relative ordering must be satisfied. However, independent sets of instructions may be effectively executed out of sequence.

In Table 1 is set forth an example of a SESE basic block representing the inner loop of a matrix multiply routine. This example will be used throughout this specification and the teachings of the present invention are application for any routine. The instruction is set forth in the left hand column and the conventional object code function for this basic block is represented in the right hand column.

TABLE 1

INSTRUCTION	OBJECT CODE
LD R0, (R10) +	I0
LD R1, (R11) +	I1
MM R0, R1, R2	I2
ADD R2, R3, R3	I3
DEC R4	I4
BRNZR LOOP	I5

The instruction stream contained within the SESE basic block set forth in Table 1 performs the following functions. In instruction I0, register R0 is loaded with the contents of memory whose address is contained in R10. The instruction shown above increments the contents of R10 after the address has been fetched from R10. The same statement can be made for instruction I1, with the exception that register R1 is loaded and register R11 incremented. The contents of register R0 are then multiplied with the contents of register R1 and stored in register R2 in instruction I2. In instruction I3, the contents of register R2 and register R3 are added together and stored in register R3. In instruction I4, register R4 is decremented. Note that instructions I2, I3 and I4 also generate a set of condition codes that reflect the status of their respective execution. In instruction I5, the contents of register R4 are indirectly tested for zero (via the condition codes generated by instruction I4). A branch occurs if the decrement operation produced a non-zero value; otherwise execution proceeds with the next basic block's first instruction.

As shown in FIG. 1, the first function performed by TOLL **110** is to analyze the resource usage of the instructions; which in Table 1 are instructions I0 through I5. The TOLL software **110** analyzes each instruction to ascertain the resource requirements of each instruction.

This analysis is important in determining whether or not sets of resources share any elements and, therefore, whether or not the instructions are independent of one another. Clearly, mutually independent instructions can be executed in parallel and are termed naturally concurrent." Instructions that are independent can be executed in parallel and do not rely on one another for any information nor do they share any hardware resources in other than a read only manner.

On the other hand, instructions that are dependent on one another can be formed into a set wherein each instruction in the set is dependent on every other instruction in that set, although this dependency may not be direct. The set can be described by the instructions within that set, or conversely, by the resources used by the instructions in that set. Instructions within different sets are completely independent of one another, i.e., there are no resources shared by the sets. Hence, the sets are independent of one another.

5,517,628

17

In the example of Table 1, there are two independent sets of dependent instructions determined by TOLL:

Set 1:	CC1:	I0, I1, I2, I3
		and
Set 2:	CC2:	I4, I5

As can be seen, instructions I4 and I5 are independent of instructions I0–I3. In set 2, I5 is directly dependent on I4. In set 1, I2 is directly dependent on I0 and I1. Instruction I3 is directly dependent on I2 and indirectly dependent on I0 and I1.

The TOLL software of the present invention detects these independent sets of dependent instructions and assigns condition codes sets such as CC1 and CC2 to each set. This avoids the operational dependency that would occur if only one set of condition codes were available to the instruction stream.

In other words, the results of the execution of instructions I0 and I1 are needed for the execution of instruction I2. Similarly, the results of the execution of instruction I2 are needed for the execution of instruction I3. Thus, the TOLL software 110 determines if an instruction will perform a read and/or a write to a resource. This functionality is termed the resource requirement analysis of the instruction stream.

It should be noted that, unlike the teachings of prior art, the present invention teaches that it is not necessary for dependent instructions to execute on the same processor element. The determination of dependencies is needed only to assign condition code sets and to assign instruction firing times, as will be described later. The present invention can execute dependent instructions on different processor elements because of the context free nature of the processor elements and the total coupling of the processor elements to the shared resources, such as register files, as will also be described later.

The results of the analysis stage 120, for the example set forth in Table 1, are set forth in Table 2.

TABLE 2

BB	FUNCTION
I0	Memory Read, Reg. Write, Reg. Read & Write
I1	Memory Read, Reg. Write, Reg. Read & write
I2	Two Reg. Reads, Reg. Write, Set Cond. Code (Set #1)
I3	Two Reg. Reads, Reg. Write, Set Cond. Code (Set #1)
I4	Read Reg., Reg. Write, Set Cond. Code (Set #2)
I5	Read Cond. Code (Set #2)

In Table 2, for instructions I0 and I1, a register is read and written followed by a memory read (at a distinct address), followed by a register write. Likewise, condition code writes and register reads and writes occur for instructions I2 through I4. Finally, instruction I5 is a simple read of a condition code storage and a resulting branch or loop.

The second pass 130 through the SESE basic block 100 is to add or extend the intelligence of each instruction within the basic block. This is the assignment of an instruction's execution time relative to the execution times of the other instructions in the stream, the assignment of a logical processor number on which the instruction is to execute and the assignment of any static shared context storage mapping information that may be needed.

In order to assign the instruction's firing time, the temporal usage of each resource required by the instruction must be considered. The temporal usage of each resource is characterized by a "free time" and a "load time." The free time is the last time the resource is used by an instruction.

18

The load time is the last time the resource is modified by an instruction. If an instruction is going to modify a resource, it must execute after the last time the resource is used, in other words, after the free time. If an instruction is going to read the resource, it must perform the read after the last time the resource has been loaded, in other words, after the load time.

The link between the temporal usage of each resource and the actual usage of the resource is as follows. If the instruction is going to write/modify the resource, the last time the resource is read or written by other instructions (i.e., the "free time" for the resource) plus one time interval will be the firing time for this resource. The "plus one time interval" comes from the fact that an instruction is still using the resource during the free time. On the other hand, if the instruction reads a resource, the last time the resource is modified by other instructions (i.e., the load time for the resource) plus one time interval will be the resource firing time. The "plus one time interval" comes from the time required for the instruction that is performing the load to execute.

The above discussion assumes that the exact location of the resource that is accessed is known. This is always true of resources that are directly named such as registers and condition codes. However, memory operations may, in general, be to unknown (at compile time) locations. In particular, addresses that are generated by effective addressing constructs fall under this domain. In the previous example, it has been assumed (for the purposes of communicating the basic concepts of TOLL) that the addresses used by instructions I0 and I1 are distinct. If this were not the case, the TOLL software would assure that only those instructions that did not use memory would be allowed to execute in parallel with an instruction that was accessing an unknown location in memory.

The resource firing time is evaluated by TOLL 110 for each resource that the instruction uses. These resource firing times are then compared to determine which is the largest or latest with this maximum value determining the actual firing time assigned to the instruction. At this point, TOLL 110 then updates all resources' free and load times, to reflect this instruction's firing time. TOLL 110 then proceeds by analyzing the next instruction.

There are many methods available for determining inter-instruction dependencies within a basic block. The previous discussion is just one possible implementation assuming a specific compiler-TOLL partitioning. Many other compiler-TOLL partitionings and methods for determining inter-instruction dependencies may be possible and realizable to one skilled in the art. As an example, the TOLL software uses a linked list analysis to represent the data dependencies within a basic block. Other possible data structures that could be used are trees, stacks, etc.

Assume a linked list representation is desired for the analysis and representation of the inter-instruction dependencies. Each register is associated with a set of pointers to the instructions that use the value contained in that register. For the matrix multiply example in Table 1, the resource usage is set forth in the following:

TABLE 3

Resource	Loaded By	Read By
R0	I0	I2
R1	I1	I2
R2	I2	I3

5,517,628

| 19 | 20 |

TABLE 3-continued

Resource	Loaded By	Read By
R3	I3	I3, I2
R4	I4	I5
R10	I0	I0
R11	I1	I1

Thus, by following the "read by" links and knowing the resource utilization for each instruction, the independencies of Sets 1 and 2, above, are constructed in the analyze instruction stage **120** by TOLL **110**.

For purposes of the example of Table 1, it is assumed that the basic block commences with an arbitrary time interval, in an instruction stream, such as time interval, for example purposes only, **T16**. In other words, this particular basic block in time sequence is assumed to start with time interval **T16**. The results of the analysis in stage **120** are set forth in Table 4.

TABLE 4

REG	I0	I1	I2	I3	I4	I5
R0	T16		T17			
R1		T16	T17			
R2			T17	T18		
R3				T18		
R4					T16	
CC1			T17	T18		
CC2						T17
R10	T16					
R11		T16				

The left hand column of Table 4 relates to the identity of the register or condition code storage whereas the rows in the table represent the instructions in the basic block example of Table 1. Instruction I0 requires that a register be read and written and another register written at time **T16**, the start of the basic block. Hence, at time **T16**, register **R10** is read from and written into and register **R0** is written into.

Under the teachings of the present invention, there is no reason that registers **R1, R11,** and **R4** cannot also have operations performed on them during time **T16**. These three instructions, I0, I1, and I4, are data independent of each other and can be operated on concurrently during time **T16**. Instruction I2 requires first that registers **R0** and **R1** be operated on so that they may be multiplied together and the results stored in register **R2**. Although, register **R2** could be operated on in time **T16**, instruction I2 is data dependent and depends upon the results of loading registers **R0** and **R1**, which occurs during time **T16**. Therefore, the completion of instruction I2 is data dependent and must occur during or after timeframe **T17**. Hence, in Table 4 above, the entry **T17** for the intersection of instruction I2 and register **R2** is underlined because it is data dependent. Likewise, instruction I3 requires data to be present in register **R2** which first occurs during time **T17**. Hence, register **R2** can have an operation occur on it only during or after time **T18**. As it turns out, instruction I5 depends upon the reading of the condition code storage **CC2** which is updated in instruction I4. The reading of the condition code storage is data dependent upon the results stored in time **T16** and, therefore, must occur during or after the next time, **T17**.

Hence, in stage **130**, the object code instructions are assigned "instruction firing times" (IFTs) as set forth in Table 5 based upon the above analysis.

TABLE 5

OBJECT CODE	INSTRUCTION FIRING TIME (IFT)
I0	T16
I1	T16
I2	T17
I3	T18
I4	T16
I5	T17

Each of the instructions in the sequential instruction stream in a basic block can be performed in the assigned time intervals. As is clear in Table 5, the same six instructions of Table 1 normally processed sequentially in six cycles can be processed, under the teachings of the present invention, in only three firing times: **T16, T17,** and **T18**. The instruction firing time (IFT) provides the "time-driven" feature of the present invention.

The next function in the extend intelligence stage **130** is to reorder the natural concurrencies in the instruction stream according to instruction firing times (IFTs) and then to assign the individual logical parallel processors. It should be noted that the reordering is only required due to limitations in currently available technology. If true fully associative memories were available, the reordering of the stream would not be required and the processor numbers could be assigned in a first come, first served manner. The hardware of the instruction selection mechanism could be appropriately modified by one skilled in the art to address this mode of operation.

For example, assuming currently available technology, and a system with four parallel processor elements (PEs) and a branch execution unit (BEU) within each LRD, the processor elements and the branch execution unit can be assigned, under the teachings of the present invention, according to that set forth in Table 6 below. It should be noted that the processor elements execute all non-branch instructions, while the branch execution unit (BEU) of the present invention executes all branch instructions. These will be described in greater detail subsequently.

TABLE 6

Logical Processor Number	T16	T17	T18
0	I0	I2	I3
1	I1	—	—
2	I4	—	—
4	—	—	—
BEU	—	I5 (delay)	—

Hence, under the teachings of the present invention, during time interval **T16**, parallel processor elements **0, 1,** and **2** concurrently process instructions I0, I1, and I4. Likewise, during the next time interval **T17**, parallel processor elements **0** and the user's BEU concurrently process instructions I2 and I5. And finally, during time interval **T18**, processor element **0** processes instruction I3. During instruction firing times **T16, T17,** and **T18**, parallel processor element **3** is not utilized in the example of Table 1. In actuality, since the last instruction is a branch instruction, the branch cannot occur until the last processing is finished in time **T18** for instruction I3. A delay field is built into the processing of instruction I5 so that even though it is processed in time interval **T17**, its execution is delayed before looping or branching out until after instruction I3 has executed.

The TOLL software **110** of the present invention in the extend intelligence stage **130** looks at each individual

5,517,628

21

instruction and its resource usage both as to type and as to location (if known) (e.g., Table 3). It then assigns instruction firing times (IFTs) on the basis of this resource usage (e.g., Table 4), reorders the instruction stream based upon these firing times (e.g., Table 5) and assigns logical processor numbers (LPNs) (e.g., Table 6) as a result thereof.

The extended intelligence information involving the logical processor number (LPN) and the instruction firing time (IFT) is added to each instruction of the basic block as shown in FIGS. 3 and 4. As will also be pointed out subsequently, this extended intelligence (EXT) for each instruction in a basic block (BB) will be translated onto the physical processor architecture of the present invention. The physical translation is done by hardware. It is important to note that the actual hardware may contain less, the same as, or more physical processor elements than the number of logical processor elements.

The Shared Context Storage Mapping (SCSM) information shown in FIG. 4 and attached to an instruction has two components, static and dynamic. Static information is attached by the TOLL software or compiler and is a result of the static analysis of the instruction stream. Dynamic information is attached at execution time by a logical resource drive (LRD) as will be discussed later.

At this stage 130, the TOLL software 110 has analyzed the instruction stream as a set of single entry single exit (SESE) basic blocks (BBs) for natural concurrencies that can be processed individually by separate processor elements (PEs) and has assigned to each instruction an instruction firing time (IFT) and a logical processor number (LPN). Under the teachings of the present invention, the instruction stream is pre-processed by TOLL to statically allocate all processing resources in advance of execution. This is done once for any given program and is applicable to any one of a number of different program languages such as FORTRAN, COBOL, PASCAL, BASIC, etc.

In stage 140, the TOLL software 110 builds execution sets (ESs). These are set forth in FIG. 5 wherein a series of basic blocks (BBs) form a single execution set (ES). Once TOLL identifies an execution set 500, header 510 and/or trailer 520 information is placed on the ends. In the preferred embodiment only header information 510 is attached although the invention is not so limited.

Under the teachings of the present invention, basic blocks generally follow one another in the instruction stream. There may be no need for re-ordering of the basic blocks even though individual instructions within a basic block, as discussed above, are re-ordered and assigned extended intelligence information. However, the invention is not so limited. Each basic block is single entry and single exit (SESE) with the exit through a branch instruction. Typically, the branch to another instruction is within a localized neighborhood such as within 400 instructions of the branch. The purpose of forming the execution sets (stage 140) is to determine the minimum number of basic blocks that can exist within an execution set such that the number of "instruction cache faults" are minimized. In other words, in a given execution set, branches or transfers out of an execution set are statistically minimized. TOLL in stage 140 can use a number of conventional techniques for solving this linear programming-like problem which is based upon branch distances and the like. The purpose is to define an execution set as set forth in FIG. 5 so that the execution set can be placed in a hardware cache, as will be discussed subsequently, in order to minimize instruction cache faults (i.e., transfers out of the execution set).

What has been set forth above is an example shown in Tables 1 through 6 of TOLL software 110 in a single context

22

of use. In essence, TOLL determines the natural concurrencies within the instruction streams for each basic block within a given program. TOLL adds an instruction firing time (IFT) and a logical processor number (LPN) to each instruction in the determined natural concurrencies. Hence, all processing resources are statically allocated in advance of processing. The TOLL software of the present invention can be used in a number of different programs, each being used by the same or different users on a processing system of the present invention as will be explained next.

3. General Hardware Description

In FIG. 6, the block diagram format of the system architecture of the present invention termed "TDA" is shown. The TDA system architecture 600 includes a memory sub-system 610 interconnected to a number of logical resource drivers (LRDs) 620 over a network 630. The logical resource drivers 620 are further interconnected to a group of context free processor elements 640 over a network 650. Finally, the group of processor elements 640 are connected to the shared resources containing a pool of register set and condition code set files 660 over a network 670. The LRD-memory network 630, the PE-LRD network 650, and the PE-context file network 670 are full access networks that could be composed of conventional crossbar networks, omega networks, banyon networks, or the like. The networks are full access (non-blocking in space) so that, for example, any processor element 640 can access any register file or condition code file in any context 660. Likewise, any processor element 640 can access any logical resource driver 620 and any logical resource driver 620 can access any portion of the memory subsystem 610. In addition, the PE-LRD and PE-context networks are non-blocking in time. In other words, these two networks guarantee access to any resource from any resource regardless of load conditions on the network. The architecture of the switching elements of the PE-LRD network 650 and the PE-context network 670 are considerably simplified since the TOLL software guarantees that collisions in the network will never occur. The diagram of FIG. 6 is a MIMD system wherein one context 660 corresponds to at least one user program.

The memory subsystem 610 can be constructed using a conventional memory architecture and conventional memory elements. There are many such architectures and elements that could be constructed by a person skilled in the art that would satisfy the requirements of this system. For example, a banked memory architecture could be used. *High Speed Memory Systems*, A. V. Pohm and O. P. Agrawal, Beston Publishing Co., 1983.

The logical resource drivers 620 are unique to the system architecture 600 of the present invention. Each LRD provides the data cache and instruction selection support for a single user (who is assigned a context) on a timeshared basis. The LRDs receive execution sets from the various users wherein one or more execution sets per context is stored on any given LRD. The instructions within the basic blocks of the stored execution sets are stored in queues based on the logical processor number. For example, if the system has 64 users and 8 LRDs, 8 users would share an individual LRD on a timeshared basis. The operating system determines who gets an individual LRD and for how long. The LRD is detailed at length subsequently.

The context free processor elements 640 are also unique to the TDA system architecture and will be discussed later. These processor elements display the context free stochastic property in which the future state of the system depends only on the present state of the system and not on the path by which the present state was achieved. As such, architectur-